NST 2.1 User Manual

July 1, 2010

This document is the manual and users' guide to the 2.1.*x* series of the NST test framework. NST is a unit test system for Common Lisp which provides support for test fixture data, stateful setup and cleanup of tests, grouping of tests, and (we think!) a useful runtime interface. Suggestions and comments are welcome; a list of known bugs and infelicities concludes this document. The files in the NST distribution's `self-test` directory, especially `self-test/core/builtin-checks.lisp`, holds the NST tests for NST and contain many examples (some of which we have adapted for this manual).

Contents

Contributors. The primarily author of both NST and this manual is John Maraist[1]. Robert P. Goldman provided guidance, comments and suggestions through the development. Other contributors include Mike Pelican, Steve Harp and Michael Atighetchi.

[1]Smart Information Flow Technologies, 211 North First Street, Suite 300, Minneapolis, MN 55401; *jmaraist* at *sift.info*.

1 Fixtures

Fixtures are data structures and values which may be referred to by name during testing. NST provides the ability to use fixtures across multiple tests and test groups, and to inject fixtures into the runtime namespace for debugging. A set of fixtures is defined using the `def-fixtures` macro:

```
(def-fixtures FIXTURE-NAME
        ([ :uses USES ]
         [ :assumes ASSUMES ]
         [ :outer OUTER ]
         [ :inner INNER ]
         [ :documentation DOCUMENTATION ]
         [ :cache FLAG ]
         [ :export-names FLAG ]
         [ :export-fixture-name FLAG ]
         [ :export-bound-names FLAG ])
  ([ ([ :cache FLAG ]) ] NAME FORM)
  ([ ([ :cache FLAG ]) ] NAME FORM)
  ...
  ([ ([ :cache FLAG ]) ] NAME FORM))
```

`FIXTURE-NAME` is the name to be associated with this set of fixtures. `USES` is a list of the names of other fixture sets which this declaration assumes to be available. This declaration is optional, but will supress some warnings. `OUTER` and `INNER` are lists of declarations to be included in a `declare` statement respectively outside and inside of the `let`-binding in which the `BINDINGS` are applied. `DOCUMENTATION` describes the fixture set.

When a fixture is attached to a test or test group, each `NAME` defined in that fixture becomes available in the body of that test or group as if `let*`-bound to the corresponding `FORM`. A fixture in one set may refer back to other fixtures in the same set (again *à la* `let*`) but forward references are not allowed.

The names of a fixture and the names it binds can be exported from the package where the fixture is defined using the `export-bound-names` and `export-fixture-name` arguments. The default value of both is `nil` unless a non-`nil` value is provided for `export-names`.

The `cache` option, if non-nil, directs NST to evaluate a fixture's form one single time, and re-use the resulting value on subsequent applications of the fixture. Note that if this value is mutated by the test cases, test behavior may become unpredictable! However this option can considerably improve performance when constant-valued fixtures are applied repeatedly. Caching may be set on or off (the default is off) for the entire fixture set, and the setting may vary for individual fixtures.

Examples of fixture definitions:

```
(def-fixtures f1 ()
  (c 3)
  (d 'asdfg))
(def-fixtures f2 (:uses (f1))
  (d 4)
  (e 'asdfg)
  (f c))
(def-fixtures f3 ()
  ((:cache t)   g (ackermann 1 2))
  ((:cache nil) h (factorial 5)))
```

To cause a side-effect among the evaluation of a fixture's name definitions, `nil` can be provided as a fixture name. In uses of the fixture, NST will replace `nil` with a non-interned symbol; in documentation string such as for `:whatis`, any `nil`s are omitted.

The `with-fixtures` macro faciliates debugging and other non-NST uses of fixtures sets:

```
(with-fixtures (FIXTURE ... FIXTURE)
  FORM
  ...
  FORM)
```

This macro evaluates the forms in a namespace expanded with the bindings provided by the fixtures.

2 Test groups

Groups of tests can be associated with fixture sets, stateful initiatization, and stateful cleanup. The syntax of a test group declaration is:

```
(def-test-group NAME (FIXTURE FIXTURE ... FIXTURE)
  [ (:setup FORM FORM ... FORM) ]
  [ (:cleanup FORM FORM ... FORM) ]
  [ (:fixtures-setup FORM FORM ... FORM) ]
  [ (:fixtures-cleanup FORM FORM ... FORM) ]
  [ (:each-setup FORM FORM ... FORM) ]
  [ (:each-cleanup FORM FORM ... FORM) ]
  [ (:documentation STRING) ]
  TEST
  TEST
  ...
  TEST)
```

NAME is the name of this test group. The FIXTUREs are to be applied to the tests in this group.

The :setup forms are run after inclusion of names from fixture sets, but before any tests from the group. Individual tests should make no assumptions as to whether the setup is unique to that test, or whether it is shared among several tests of that group.

The :cleanup forms are normally run after the setup completes; however the cleanup form will not be run if the setup form raises an error. When the user asks the runtime system to enter the debugger on an error, the cleanup form will not run unless the user explicitly enabled a resumption of the test routine from the debugger. The cleanup form will be run in other circumstances, including at a user-requested break in testing at either failure or error.

The :fixtures-setup (respectively :fixtures-cleanup) form is run before fixtures are bound (after their bindings are released). These forms are useful, for example, to initialize a database connection from which the fixture values are drawn.

The :each-setup and :each-cleanup forms are run before each test, rather than once for the group.

The :documentation form sets the docstring for the class.

3 Tests and test criteria

Individual unit tests are encoded with the def-test form:

```
(def-test (NAME [ :group GROUP-NAME ]
                [ :setup FORM ]
                [ :cleanup FORM ]
                [ :fixtures (FIXTURE FIXTURE ... FIXTURE) ]
                [ :documentation STRING ] )
    criterion
  FORM FORM ... FORM)

(def-test NAME
    criterion
  FORM FORM ... FORM)
```

The SETUP, CLEANUP and FIXTURES are just as for test groups, but apply only to the one test. The CRITERION is a list or symbol specifying the properties which should hold for the FORMs.

When a test is not enclosed within a group body, a group name must be provided by the GROUP option. When a test is enclosed within a group body, the GROUP

option is not required, but if provided it must agree with the group name.

When there are no `SETUP`, `CLEANUP` or `FIXTURES` arguments, the `NAME` may be given without parentheses. Likewise, any criterion consisting of a single symbol, e.g. `(:pass)`, may be abbreviated as just the symbol without the parentheses, e.g. `:pass`.

The `:documentation` form provides a documentation string in the standard Lisp sense. Since documentation strings are stored against names, and since the same name can be used for several tests (so long as they are all in different packages), documentation strings on tests may not be particularly useful.

The `def-check` form is a deprecated synonym for `def-test`.

3.1 Basic criteria

3.1.1 The `:true` criterion

The form is evaluated at testing time; the criterion requires the result to be non-nil.
Syntax: `:true`

3.1.2 The `:eq` criterion

The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be `eq`.
Syntax: `(:eq FORM)`
Example: `(def-test eq1 (:eq 'b) (cadr '(a b c)))`

3.1.3 The `:symbol` criterion

The form under test is evaluated at testing time. The criterion requires that the result be a symbol which is `eq` to the symbol name given as the criterion argument.
Syntax: `(:symbol NAME)`
Passing example: `(def-test sym1 (:symbol a) (car '(a b c)))`
Failing example: `(def-test sym1x (:symbol a) (cadr '(a b c)))`

3.1.4 The `:eql` criterion

The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be `eql`.
Syntax: `(:eql FORM)`
Example: `(def-test eql1 (:eql 2) (cadr '(1 2 3)))`

3.1.5 The `:equal` criterion

The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be `equal`.
Syntax: `(:equal FORM)`

3.1.6 The `:equalp` criterion

The criterion argument and the form under test are both evaluated at testing time; the criterion requires that the results be `equalp`.
Syntax: `(:equalp FORM)`

3.1.7 The `:forms-eq` criterion

The two forms under test are both evaluated at testing time; the criterion requires that the results be `eq`.
Syntax: `:forms-eq`
Example: `(def-test eqforms1 :forms-eq (cadr '(a b c)) (caddr '(a c b)))`

3.1.8 The `:forms-eql` criterion

The two forms under test are both evaluated at testing time; the criterion requires that the results be `eql`.
Syntax: `:forms-eql`
Example: `(def-test eqlforms1 :forms-eql (cadr '(a 3 c)) (caddr '(a c 3)))`

3.1.9 The `:forms-equal` criterion

The two forms under test are both evaluated at testing time; the criterion requires that the results be `equal`.
Syntax: `:forms-equal`

3.1.10 The `:predicate` criterion

The criterion argument is a symbol (unquoted) or a lambda expression; at testing time, the forms under test are evaluated and passed to the denoted function. The criterion expects that the result of the function is non-nil.
Syntax: `(:predicate FUNCTION-FORM)`
Passing example: `(def-test pred1 (:predicate numberp) 3)`
Passing example: `(def-test pred2 (:predicate eql) (+ 1 2) 3)`

3.1.11 The `:err` criterion

At testing time, evaluates the form under test, expecting the evaluation to raise some condition. If the *CLASS* argument is supplied, the criterion expects the raised condition to be a subclass. Note that the name of the type should *not* be quoted; it is not evaluated.
Syntax: `(:err [:type CLASS])`
Passing example: `(def-test err1 (:err :type error) (error "this should be caught"))`
Passing example: `(def-test err2 (:err) (error "this should be caught"))`

3.1.12 The `:perf` criterion

Evaluates the forms under test at testing time, and expects the evaluation to complete within the given time limit.
Syntax: `(:perf [ :ns | :sec | :min ] TIME)`
Example: `(def-test perf1 (:perf :min 2) (ack 3 5))`

3.2 Compound criteria

3.2.1 The `:not` criterion

Passes when testing according to `CRITERION` fails (but does not throw an error).
Syntax: `(:not CRITERION)`
Example: `(def-test not1 (:not (:symbol b)) 'a)`

3.2.2 The `:all` criterion

This criterion brings several other criteria under one check, and verifies that they all pass.
Syntax: `(:all CRITERION CRITERION ... CRITERION)`
Example:

```
(def-check not1 ()
     (:all (:predicate even-p)
           (:predicate prime-p))
   2)
```

3.2.3 The `:any` criterion

Passes when any of the subordinate criteria pass.
Syntax: `(:any CRITERION CRITERION ... CRITERION)`
Example:

```
(def-check not1 ()
     (:any (:predicate even-p)
           (:predicate prime-p))
   5)
```

3.2.4 The `:apply` criterion

At testing time, first evaluates the forms under test, applying `FUNCTION` to them. The overall criterion passes or fails exactly when the subordinate `CRITERION` with the application's multiple result values.
Syntax: `(:apply FUNCTION CRITERION)`
Example: `(def-test applycheck (:apply cadr (:eql 10)) '(0 10 20))`

3.2.5 The `:check-err` criterion

Like `:err`, but proceeds according to the subordinate criterion rather than simply evaluating the input forms.
Syntax: `(:check-err CRITERION)`
Example:

```
(def-test check-err1
     (:check-err :forms-eq)
   'asdfgh (error "this should be caught"))
```

The difference between `:check-err` and `:err` is that the latter deals only with evaluation of a form, whereas `:check-err` is more about the unit testing process. This form is mostly useful for temporarily disregarding certain checks until some later fix, when they *won't* throw an error.

3.2.6 The `:progn` criterion

At testing time, first evaluates the `FORM`s in order, and then proceeds with evaluation of the forms under test according to the subordinate criterion.
Syntax: `(:progn FORM FORM ... FORM CRITERION)`
Example: `(def-test form1 (:progn (setf zz 3) (:eql 3)) zz)`

3.2.7 The `:proj` criterion

Rearranges the forms under test by selecting a new list according to the index numbers into the old list. Checking of the reorganized forms continues according to the subordinate criterion.
Syntax: `(:proj (INDEX INDEX ... INDEX) CRITERION)`
Example:

```
(def-test proj-1
    (:proj (0 2) :forms-eq)
  'a 3 (car '(a b)))
```

Note that containing criteria may have reordered forms and value from the original check.

3.3 Criteria for multiple values

NST's approach to multiple values is stricter than Common Lisp's view in the language itself. In Lisp programs, additional returned values are eminently ignorable; in fact some extra programming overhead is required to access them. However, in designing NST we draw a distinction between *using* a function on the one hand, and *designing* and *testing* it on the other. For example, when using the `floor` function one often needs only the quotient, and it is correspondingly easy to ignore the additional argument which gives the remainder. However, for the implementor of `floor` it is impormtant that both results be predictable and verified.

NST encourages the thoughtful, consistent design and correct implementation of functions returning multiple values by enforcing that tested forms generating multiple variables should be paired with criteria for multiple values. *Any mismatch between the quantity of values returned by an evaluation and the quantity of values expected by a criterion is interpreted as a test failure.* Dually to the usage mode of Common Lisp, NST requires the additional overhead of the `:drop-values` criterion to simply ignore additional returned values.

3.3.1 The `:value-list` criterion

Converts multiple values into a single list value.
Syntax: `(:value-list CRITERION)`

3.3.2 The `:values` criterion

Checks each of the forms under test according to the respective subordinate criterion.
Syntax: `(:values CRITERION CRITERION ... CRITERION)`

3.3.3 The `:drop-values` criterion

Checks the primary value according to the subordinate criterion, ignoring any additional returned values from the evaluation of the form under test.
Syntax: `(:drop-values CRITERION)`

3.4 Criteria for lists

3.4.1 The `:each` criterion

At testing time, evaluates the form under test, expecting to find a list as a result. Expects that each argument of the list according to the subordinate `CRITERION`, and passes when all of these checks pass.
Syntax: `(:each CRITERION)`
Example: `(def-test each1 (:each (:symbol a)) '(a a a a))`

3.4.2 The `:seq` criterion

Evaluates its input form, checks each of its elements according to the respective subordinate criterion, and passes when all of them pass.
Syntax: `(:values CRITERION CRITERION ... CRITERION)`
Example:

```
(def-check seqcheck
     (:seq (:predicate symbolp) (:eql 1) (:symbol d))
   '(a 1 d))
```

Note that `:seq` expects that the length of the list will be the same as the number of subordinate criteria, and will fail otherwise.

3.4.3 The `:permute` criterion

At testing time, evaluates the form under test, expecting to find a list as a result. The criterion expects to find that some permutation of this list will satisfy the subordinate criterion.
Syntax: `(:permute CRITERION)`
Example:

```
Examples: (def-test permute1 (:permute (:each (:eq 'a))) '(a a))
          (def-check permute2
               (:permute (:seq (:symbol b)
                               (:predicate symbolp)
                               (:predicate numberp)))
            '(1 a b))
```

3.5 Criteria for vectors

3.5.1 The `:across` criterion

Like `:seq`, but for a vector instead of a list.
Syntax: `(:across CRITERION CRITERION ... CRITERION)`
Example:

```
(def-check across1
     (:across (:predicate symbolp) (:eql 1))
   (vector 'a 1))
```

3.6 Criteria for classes

3.6.1 The `:slots` criterion

Evaluates its input form, and passes when the value at each given slot satisfies the corresponding subordinate constraint.
Syntax: `(:slots (NAME CRT) (NAME CRT) ... (NAME CRT))`
Example:

```
(defclass classcheck ()
     ((s1 :initarg :s1 :reader get-s1)
      (s2 :initarg :s2)
      (s3 :initarg :s3)))
(def-test slot1
          (:slots (s1 (:eql 10))
                  (s2 (:symbol zz))
                  (s3 (:seq (:symbol q) (:symbol w)
                            (:symbol e) (:symbol r))))
          (make-instance 'classcheck
             :s1 10 :s2 'zz :s3 '(q w e r)))
```

Use of this criterion with structs rather than classes does work on many platforms, since the CL specification defines `with-slots` on classes only.

3.7 Special criteria

3.7.1 The `:sample` criterion

Experimentally test a program property by generating random data. See Section 5 for more information.
Syntax: `(:sample &key domains where verify values sample-size qualifying-sample max-tries)`

3.8 Programmatic and debugging criteria

3.8.1 The `:info` criterion

Add an informational note to the check result.
Syntax: `(:info MESSAGE SUBCRITERION)`
Example: `(def-test known-bug (:info "Known bug" (:eql 3)) 4)`

3.8.2 The `:pass` criterion

A trivial test, which always passes.
Syntax: `:pass`
Example: `(def-test passing-test :pass 3 4 "sd")`

3.8.3 The `:fail` criterion

A trivial test, which always fails. The format string and arguments should be suitable for the Lisp `format` function.
Syntax: `(:fail FORMAT ARG ... ARG)`
Example: `(def-test fails (:fail "Expected a ~a" "string") 312)`

3.8.4 The `:warn` criterion

Issue a warning. The format string and arguments should be suitable for the Lisp `format` function.
Syntax: `(:warn FORMAT ARG ... ARG)`
Example: `(:warn "~d is not a perfect square" 5)`

3.8.5 The `:dump-forms` criterion

For debugging NST criteria: fails after writes the current forms to standard output.
Syntax: `(:dump-forms FORMAT)`

4 Defining test criteria

The criteria used in test forms decide whether, when and how to use the forms under test and the forms and subcriteria provided to each test criterion. Criteria receive their arguments as forms, and may examine them as forms with or without evaluation, as the particular criterion requires. NST provides three mechanisms for defining new criteria.

- Defining a criterion by specifying how it should be rewritten to another criterion. This mechanism is both the simplest and the most limited in the manipulations it can define. The `def-criterion-alias` macro provides this mechanism, which we discuss in Section 4.1.
- Defining a criterion with call-by-value semantics for the values under test, specifying how it assesses the results of evaluating the forms under test. The `def-criterion` macro provides this mechanism, which we discuss in Section 4.3.
- Defining a criterion receiving the original, unmanipulated forms provided as criterion arguments and forms under test. The `def-criterion-unevaluated` macro provides this mechanism, which we discuss in Section 4.5.

The first mechanism is essentially a variation of `defmacro`. Under both of the latter two mechanisms, the criteria definition is made as Lisp code calculating a *test report*.

The functions and macros for defining new criteria are exported from package `nst-criteria-api`.

4.1 Aliases over criteria

The simplest mechanism for defining a new criterion involves simply defining one criterion to rewrite as another using `def-criterion-alias`:

Syntax:
```
(def-criterion-alias (name &rest args)
          [ documentation ]
          expansion)
```

The body of the expansion should be a Lisp form which, when evaluated, returns an S-expression quoting the new criterion which the rewrite should produce. The `args` are passed as far Lisp macros: they are not evaluated and are most typically comma-inserted into a backquoted result. For example:
```
(def-criterion-alias (:forms-eq) `(:predicate eq))
(def-criterion-alias (:symbol name) `(:eq ',name))
```

4.2 Reporting forms

The other two criteria-defining mechanisms define the expansion of a criterion into Lisp. For both of these mechanisms, this Lisp code is expected to return a test report. NST provides three functions for building test reports:

- `(make-success-report)`

  This function indicates a successful test result.

Note that some older examples show `(make-check-result)`, `(emit-success)` or `(check-result)`. The former is an internal function and should not be used from outside the core NST files. The latter two are deprecated.

- `(make-failure-report [ :format format-string [ :args args ] ] )`

  This function returns a report of test failure. The `format-string` and `args` are as to the Common Lisp function `format`. The `emit-failure` function is an older, deprecated version of this function.

- `(make-warning-report [ :format format-string [ :args args ] ] )`

  Like `make-failure-report`, but provides supplimentary information as a warning. The `emit-warning` function is an older, deprecated version of this function.

4.3 Defining criteria over evaluated values

Syntax:
```
(def-criterion (name criterion-body-lambda-list
                     actual-values-lambda-list)
        FORM
        FORM
         ⋮
        FORM)
```

Examples:

```
(def-criterion (:true () (bool))
  (if bool
      (make-success-report)
      (make-failure-report :format "Expected non-null, got: ~s"
                   :args (list bool))))

(def-criterion (:eql (target) (actual))
  (if (eql (eval target) actual)
      (make-success-report)
      (make-failure-report :format "Not eql to value of ~s"
                   :args (list target))))
```

These criteria definitions are like generic function method definitions with two sets of formal parameters:

- The forms provided as the actual parameters of the criterion itself.

- The values arising from the evaluation of the forms under test.

The body of a `def-criterion` should return a test result report contructed by the function described in Section 4.2 above.

4.4 Processing subcriteria on values

Since the arguments to the criterion itself (as opposed to the tested forms) are passed unevaluated as for macro arguments, they can contain *subcriteria* which can be incorporated into the main criterion's assessment.

Syntax: `(check-subcriterion-on-value CRITERION EXPR)`

4.5 General criteria definitions

Syntax:
```
(def-criterion-unevaluated (name
                            criterion-args-lambda-list)
                            form-argument)
    FORM
    FORM
     ⋮
    FORM)
```

As under `def-criterion`, the body of these criteria definitions receive the forms provided as the actual parameters of the criterion itself, and should return a test result report. However, these criteria receive the unevaluated forms under test, deciding when and whether to evaluate them.

4.6 Processing subcriteria on the unevaluated form

Syntax: `(check-subcriterion-on-form CRITERION FORM)`

4.7 Older criteria-defining macros

The `def-values-criterion` and `def-form-criterion` macros are deprecated as of NST 1.3.0, and will be make-emoved-report at some point. Code using `def-values- criterion` should continue to work as before. *However, code using `def-form- criterion` in any but the simplest ways is very likely to fail.* In NST 1.3 criteria are translated into method definitions, whereas in earlier versions criteria guided the macro expansion of tests. Unfortunately, the nature of `def-form-criterion` declarations eludes translation into the new scheme.

5 Verifying invariants against sampled data

The `:sample` criterion provides random generation of data for validating program properties.[2] Our approach is based on Claessen and Hughes's Quickcheck[3].

This style of testing is somewhat more complicated to than specific tests on single, bespoke forms. There are two distinct efforts, which we address in the next two sections: describing how the sample data can be generating, and specifying the test itself.

5.1 Generating sample data

Data generation is centered around the generic function `arbitrary`. This function takes a single argument, which determines the type of the value to be generated. For simple types, the name of the type (or the class object, such as returned by `find-class`) by itself is a complete specification. For more complicated types, `arbitrary` can also take a list argument, where the first element gives the type and the remaining elements are keyword argument providing additional requirements for the generated value.

NST provides method of `arbitrary` for many standard Lisp types, listed in Table 1. Types in the first column — the standard numeric types plus the common supertype `t` — are not associated with additional keyword arguments.

```
(nst:arbitrary t)
(nst:arbitrary 'complex)
(nst:arbitrary 'integer)
(nst:arbitrary 'ratio)
(nst:arbitrary 'single-float)
```

Keyword arguments for other NST-provided type specifiers are as follows:

- Types `character` and `string`:
  - Argument `noncontrol`. Excludes the control characters associated with ASCII code 0 through 31.
  - Argument `range`. Allows the range of characters to be restricted to a particular subset:

[2] This feature appears first in NST version 1.1.2.

[3] Koen Claessen and John Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," from *Proceedings of the International Conference on Functional Programming*, 2000. QuickCheck papers, code and other resources are available at *www.cs.chalmers.se/~rjmh/QuickCheck* .

[4] Not available on Allegro Lisp.

Standard Lisp types					Other types
number	character	symbol	cons	hash-table	scalar
real	string		list		
rational			vector		
integer			array		
float				t	
fixnum					
bignum					
ratio					
short-float[4]					
single-float					
double-float[4]					
long-float					
complex					
Considered scalar					

Table 1: NST provides methods of generic function `arbitrary` generating values of the types in this table.

Value	Meaning
`:standard`	Codes up to 96
`:ascii`	Codes through 127
`:ascii-ext`	Codes through 255

Omitted or with any other value, characters with any code up to `char-code-limit` can result. Examples:

```
(nst:arbitrary 'character)
(nst:arbitrary '(character :noncontrol t
                           :range :standard))
```

- Type `symbol`:
  - Argument `existing`. If non-nil, requires that the result be a previously-interned symbol.
  - Argument `exported`. Requires that the result be not only a previously-interned symbol, but also one exported by its package. Ignored if `:existing` is explicitly set to nil.
  - Argument `package`. Specifies the package from which the symbol will be generated. If omitted, a package is selected at random from the existing ones.
  - Argument `nonnull`. If non-nil, allows `arbitrary` to ignore other restriction to guarantee returning a non-nil symbol. When null, `arbitrary` may return nil.
  - Argument `gensym`. If non-nil, and if `:existing` is explicitly set to nil, returns a new uninterned symbol.

- Type `cons`:
  - Arguments `car` and `cdr` should be additional type specifications, used direct the generation of respectively the left and right elements of the result. Each defaults to `t` .
- Types `list` and `vector`:
  - Argument `length` specifies the length of the structure. If omitted, will be randomly generated.
  - Argument `elem` directs the generation of the container's elements. For both, the default element type is `t` .
- Type `array`:
  - Argument `elem`. As for `list` and `vector`.
  - Argument `dimens`. Should be a list of nonnegative integers specifying the length of each dimension of the array. If omitted, will be randomly generated.
  - Argument `rank`. Specifies the number of dimensions. If omitted but `:dimens` is given, will be set to the length of `:dimens`. If both `:rank` and `:dimens` are omitted, then both are randomly generated.
- Type `hash-table`:
  - Argument `size`. Specifies the number of entries in the table. If omitted, will be randomly generated.
  - Argument `test`. Specifies the hash table's test function. If omitted, will be randomly selected from `eq`, `eql`, `equal` and `equalp`.
  - Arguments `key` and `val` direct the generation of the table's keys and values, respectively. For the keys, the default element type is `t` when the test function is `eq` or `eql`, and `scalar` otherwise. For the values, the default element type is `t` .

Beyond those standard Lisp types, NST provides the type `scalar` as a supertype of the numeric types, `character`, `string` and `symbol`. Users may extend this definition to include additional type specifications, as we discuss below. Types are not associated with `scalar` are referred to as *compound* (although there is no corresponding type specification). To avoid generating structures too large to hold in memory, NST provides the global variable `*max-compound-structure-depth*` which sets the maximum nesting depth of compound data structures: beyond that depth, `scalar` rather than `t` is the default element generator. This restriction does not apply to explicitly specified element types, only to the use of defaults.

New type specifications are defined with the `def-arbitrary-instance-type` macro.

```
(def-arbitrary-instance-type (SPECIFICATION-NAME
                                [ :key KEYWORD-PARAMS ]
                                [ :scalar BOOL ] )
  FORM
  ...
  FORM)
```

When a non-null value is provided for the `:scalar` argument, the new specifier is taken to be generable by the `scalar` specification.

```
(def-arbitrary-instance-type (ratio :scalar t)
  (/ (arbitrary 'integer)
     (let ((raw (arbitrary (find-class 'integer))))
       (cond
         ((< raw 0) raw)
         (t (+ 1 raw))))))
```

The `:key` argument gives a list of keyword arguments which may accompany the new specification. For the `cons` type, keyword arguments allow specifications for the left and right components:

```
(def-arbitrary-instance-type (cons :key ((car t car-supp-p)
                                         (cdr t cdr-supp-p)))
  (compound-structure
   (when (and (not car-supp-p)
              (>= *current-compound-structure-depth*
                  *max-compound-structure-depth*))
     (setf car 'scalar))
   (when (and (not cdr-supp-p)
              (>= *current-compound-structure-depth*
                  *max-compound-structure-depth*))
     (setf cdr 'scalar))
   (cons (arbitrary car) (arbitrary cdr))))
```

5.2 Invariants as tests

Invariants to be tested, and the domains over which they range, are specified in the `:sample` criterion:

```
(:sample [ :value LAMBDA-LIST ]
         [ :domains ((NAME SPEC) ... (NAME SPEC)) ]
         [ :where FORM ]
         :verify FORM
         [ :sample-size NUMBER ]
```

```
        [ :qualifying-sample NUMBER ]
        [ :max-tries NUMBER ] )
```

The `verify` argument is the expression to be (repeatedly) evaluated, and which is expected always to return a non-null value. This is the sole required argument, although in any particular use it is unlikely to be the only argument given. The other arguments are:

- The `domains` argument declares the variables in the `verify` expression which are to be given multiple randomized values. The default value is `nil`, denoting an empty list.
- The `value` argument is a lambda list to which the values given by the argument form should be applied. The default value is `nil`, denoting no such arguments.
- The `where` argument is a condition which determines the validity of the input argument. For example, the condition would assert that a number is positive in an application where a negative value would be known to cause a failure. The default value is `t`, allowing any values.
- The `sample-size` argument gives the base specification of the number of value sets which will be generated. Two further arguments have some bearing on the number of generation attempts when the `where` argument is non-`t`. The `qualifying-sample` argument gives the minimum acceptable size of actual tested values, not counting sets rejected via the `where` expression. The `max-tries` argument gives the maximum number of value sets to be generated.

Examples:

```
(:sample :sample-size 10
         :domains ((x (list :elem symbol)))
         :verify (equal x (reverse (reverse x))))

(:sample :domains ((x real))
         :where (> x 1)
         :verify (< (sqrt x) x)
         :sample-size 10
         :max-tries 12)
```

6 The runtime system

The runtime system provides several operations for scheduling and running tests, and debugging failing and erring tests. The operations are accessible from the `nst-cmd` macro. Under Allegro, the top-level alias `:nst` provides a shorthand to this function; for the sake of brevity we use this shorthand below.

The `:help` command gives a complete inventory of runtime system commands:

```
:nst :help
(nst-cmd :help)
```

There are a numnber of commands for running tests, but most of the time only one will be needed:

- `:nst :run` *name*

  Run all tests in the named package, or in the named group, or run the named test. It is not necessary to prefix the name with a package prefix.

When a name corresponds to several different types of entities, or to different entities in different packages, it is necessary to use a more specific instruction:

- `:nst :run-package` *name name ... name*

  Run all tests defined in groups in the named packages. If no packages are given, then the current value of `*package*` is used.

- `:nst :run-group` *group-name*

  Run all tests in the given group. Where appropriate, the name should be package-prefixed.

- `:nst :run-test` *group-name test-name*

  Run the named test. Where appropriate, the names should be package-prefixed.

One further command for running a test is useful when writing and debugging the tests themselves:

- `:nst :apply` *criterion form form* $\cdots$ *form*

  Test the *form*s against the given *criterion*. The test proceeds just as if the criterion and forms were given in a `def-test` and that test run. Of course, any fixtures expected in one of the *form*s would not necessarily be present in the runtime environment; fixtures may need to be `open`ed.

There are two commands for (re)printing the results of tests:

- `:nst :report`
  `:nst :report` *`package-name`*
  `:nst :report` *`group-name`*
  `:nst :report` *`group-name test-name`*

- `:nst :detail`
  `:nst :detail` *`package-name`*
  `:nst :detail` *`group-name`*
  `:nst :detail` *`group-name test-name`*

The `:report` command summarizes successes, failures and errors; the `:detail` command gives more detailed information about individual tests.

The `:undef` command cancels the definition of a group or test:

```
:nst :undef group-name
:nst :undef group-name test-name
```

Currently, NST does require that the symbols passed to `:undef` be correctly package-qualified.

The `:clear` command erases NST's internal record of test results.

The `:set` and `:unset` commands adjust NST's configuration.

- `:nst :set` *`property value`*

- `:nst :unset` *`property value`*

There are currently three properties which can be manipulated by `:set`:

- `:verbose` Controls the level of output at various points of NST. Valid settings are:
  - `:silent` (aka `nil`)
  - `:quiet` (aka `:default`)
  - `:verbose` (aka `t`)
  - `:vverbose`

  The `:report` and `:detail` commands operate by setting minimum levels of verbosity.

- `:debug-on-error` When this property has a non-nil value, NST will exit into the debugger when it catches an error.

- `:debug-on-fail` When this property has a non-nil value, NST will exit into the debugger whenever a test fails. This test is useful for inspecting the environment in which a test is run. Note that both `:debug-on-error` and `:debug-on-fail` apply in the case of an error; if the latter is set but

the former is not, then the debugger will be entered after an erring test completes.

The `:debug` command is a short-cut for setting this two properties.

- `:backtraces` When this property has a non-nil value, NST attempts to capture attempts the Lisp backtrace when a test throws an error. This property is only available on platform which allow programmatic examination of backtraces, which is not standardized in Common Lisp; currently we have implemented this feature on Allegro only.

  This property has a complicated default setting. Firstly, if the symbol `'common-lisp-user::*nst-generate-backtraces*` is bound when NST loads, NST will use its value as the initial value for this property. Otherwise by default, on MacOS systems the property initializes to `nil` because of a known error on that system, but this setting can be overriden by the property `:nst-unsafe-allegro-backtraces`. Finally, if none of these issues apply, the initial value is `t`.

Fixtures can be *opened* into the interactive namespace for debugging with the `:nst :open` command:
Syntax: `:nst :open FIXTURE-NAME FIXTURE-NAME ... FIXTURE-NAME`
Example:

```
CL-USER(75): (nst:def-fixtures small-fixture ()
                (fix-var1 3)
                (fix-var2 'asdfg))
NIL
CL-USER(76): (boundp 'fix-var1)
NIL
CL-USER(77): :nst :open small-fixture
Opened fixture SMALL-FIXTURE.
CL-USER(78): fix-var1
3
CL-USER(79):
```

Fixtures can be opened into a different package than where they were first defined, but these bindings are in addition to the bindings in the original package, and are made by a symbol import to the additional package.

Calling `:nst` or `(nst-cmd)` without a command argument repeats the last test-executing command.

7 Integration with ASDF

NST's integration with ASDF is a work in progress. This section described the current integration, the ways we expect it to change, and a less-flexible and

lower-level, but likely more stable, alternative integration technique.

7.1 NST's ASDF systems

From version 1.2.2, the system `:asdf-nst` provides two classes for ASDF system definitions, `asdf:nst-test-runner` and `asdf:nst-test-holder`.

Up to NST 1.2.1 `:asdf-nst` provided a single class `asdf:nst-testable`, and in the future we plan to reunify the current two classes into a single class again. However our first implementation required NST to be loaded even when a system was *not* being tested, because we had no way to distinguish the source code associated with testing from production code. We plan to solve this problem with a new file type `nst-file` in a future version of NST. This file type would *not* be compiled or loaded for the `compile-op` or `load-op` of the system, only for its `test-op`.

7.1.1 Test-running systems

ASDF systems of the `asdf:nst-test-runner` class do not themselves contain NST declarations in their source code, but may identify other systems which do, and which should be tested as a part of testing the given system. These systems also allow local definitions of NST's configuration for the execution of their tests.

Specify that a system runs NST tests by providing `:class asdf:nst-test-runner` argument to `asdf:defsystem`. Use the `:nst-systems` argument to name the systems which house the actual unit tests:

- `:nst-systems (`*system system* $\cdots$ *system*`)`

  Specifies a list of other systems which should be tested when testing this system. These other systems do *not* otherwise need to be identified as a dependency of this system (nor, for that matter, does `:nst` itself); they will be loaded upon `test-op` if they are not yet present.

Another optional argument to an `nst-test-runner` system definition is:

- `:nst-init (`*arg-list ... arg-list*`)`

  Initializing arguments to NST, to be executed after this system is loaded. Each `arg-list` is passed as the arguments as if to a call to the `nst-cmd` macro.

- `:nst-debug-config` *form*

  NST debugging customization for this system. The `FORM` Should be an expression which, when evaluated, returns a list of keyword arguments;

note that to give the list itself, it must be explicitly quoted, *which is a change of behavior from pre-1.2.2 versions.*

- `:nst-debug-protect (`*`symbol ... symbol`*`)`

  Gives a list of variables whose values should be saved before applying any configuration changes from `:nst-debug-config`, and restored after testing.

- `:nst-push-debug-config` *`t-or-nil`*

  If non-nil, then when this system is loaded its `:nst-debug` and `:nst-debug-protect` settings will be used as NST's defaults.

7.1.2 Test-containing systems

The `asdf:nst-test-holder` class is a subclass of `nst-test-runner` for systems which are not only tested via NST, but also contains NST tests in their source code.

Specify that a system defines NST tests by providing `:class asdf:nst-test-holder` to `asdf:defsystem`. The arguments for `asdf:nst-test-runner` may be used for `asdf:nst-test-holder`, as well as the following:

- `:nst-packages (`*`package package ··· package`*`)`

  When the system is tested, all groups and tests in the named packages should be run.

- `:nst-groups ((`*`package group`*`) ··· (`*`package group`*`))`

  When the system is tested, tests in the named groups should be run. Naming the package separately from the group and test in this argument (and in the similar arguments below) allows the group to be named before its package is necessarily defined.

- `:nst-tests ((`*`package group test`*`) ··· (`*`package group test`*`))`

  When the system is tested, all the named tests should be run.

The next three arguments to an `nst-testable` system are mutually exclusive, and moreover exclude any of the above group or `:nst-systems`:

- `:nst-package` *`package`*

  When the system is tested, all groups and tests in the named package should be run.

- `:nst-group (`*`package group`*`)`

  When the system is tested, all tests in the named group should be run.

```
;; NST and its ASDF interface must be loaded
;; before we can process the defsystem form.
(asdf:oos 'asdf:load-op :asdf-nst)

(defsystem :mnst
    :class nst-test-holder
    :description "The NST test suite's self-test."
    :serial t
    :nst-systems (:masdfnst)
    :nst-groups ((:mnst-simple . g1)
                 (:mnst-simple . g1a)
                 (:mnst-simple . g1a1)
                 (:mnst-simple . core-checks))
    :depends-on (:nst)
    :in-order-to ((test-op (load-op :mnst)))
    :components ((:module "core"
                  :components ((:file "byhand")
                               (:file "builtin-checks")))))
```

Figure 1: Definitions of `nst-testable` ASDF systems.

- `:nst-test (`*`package group test`*`)`

  When the system is tested, the given test should be run.

Figure 1 gives examples of `nst-testable` ASDF system definitions.

7.2 An alternate ASDF integration technique

We plan to deprecate and then remove `asdf:nst-test-holder` and `nst-test-runner` once we have implemented a unified replacement for them. To avoid the possibility of a bit-rotted test scheme, the link between a system and its unit tests can be made explicit by providing methods for ASDF generic functions which make calls to the NST API. Specifically:

- A method of the ASDF `asdf:perform` generic function specialized to the `asdf:test-op` operation and the system in question will be executed to test a system. So an appropriate method definition would begin:

  ```
  (defmethod asdf:perform ((op asdf:test-op)
                           (sys (eql (asdf:find-system
                                      :SYSTEM-NAME))))
  ```

- NST API functions for running tests are:

  - `nst:run-package`
  - `nst:run-group`
  - `nst:run-test`

- The main NST API function for printing the results of testing is `asdf:report-multiple`. In situations where only a single package, group or test is associated with a system, one of the following function may be more convenient:
  - `nst:report-package`
  - `nst:report-group`
  - `nst:report-test`

When providing an explicit `asdf:perform` method, it is also necessary to explicitly list system dependencies to NST and to the other systems which contain the tested system's unit test definitions.

A The NST API

A.1 Primary macros

`def-fixtures` — §1, p. 2.

`def-test-group` — §2, p. 3.

`def-test` — §3, p. 4.

`def-check` — §3, p. 4 — deprecated.

`def-criterion-alias` — §4.1, p. 13.

`def-check-alias` — §4.1, p. 13 — deprecated.

`def-values-criterion` — §4.7, p. 15.

`def-value-check` — §4.7, p. 15 — deprecated.

`def-form-criterion` — §4.7, p. 15.

`def-control-check` — §4.7, p. 15 — deprecated.

A.2 Functions used in criteria definitions

`make-failure-report` — §4.2, p. 13.

`make-warning-report` — §4.2, p. 13.

`make-success-report` — §4.2, p. 13.

`emit-failure` — §4.2, p. 13.

`emit-warning` — §4.2, p. 13.

`emit-success` — §4.2, p. 13.

`add-failure` — §4.2, p. 13.

`add-error` — §4.2, p. 13.

`add-info` — §4.2, p. 13.

`check-result` — §4.2, p. 13 — deprecated.

A.3 NST control and JUnit XML output

`nst-cmd`

`nst-junit-dump` — §B, p. 29.

`junit-results-by-group` — §B, p. 29.

A.4 Programmatic control of testing and output

Note that these functions are exported from the package `:nst-control-api` — they are not intended for regular use in NST tests.

`run-package`

`run-group`

`run-test`

`report-multiple`

`report-package`

`report-group`

`report-test`

`protect-nst-config`

`apply-debug-options`

A.5 User settings

`*nst-output-stream*`

`*default-report-verbosity*`

`*debug-on-error*`

A.6 Testing randomized samples

`arbitrary` — §5.1, p. 16

`compound-structure`

`def-arbitrary-instance-type` — §5.1, p. 18.

`*max-compound-structure-depth*` — §5.1, p. 18.

A.7 Other

`protect-nst-config`

`apply-debug-options`

`with-fixtures`

B Output to JUnit

NST reports can be formatted as XML for use with JUnit, although the API for this feature is underdeveloped. The `junit-results-by-group` function aligns test groups with Java classes, and individual tests with `@Test` methods.

```
(junit-results-by-group :verbose VERBOSE
                        :dir DIR :file FILE
                        :stream STREAM
                        :if-dir-does-not-exist BOOL
                        :if-file-exists BOOL)
```

Either `:dir` and `:file` options, or the `:stream` option, but not both, should be used to specify the target for XML output; if none of the three options are given, the function will write to `*standard-output*`.

C Issues, bugs and enhancements

Tickets for NST bugs and wishes are on NST's trac:

`https://svn.sift.info:3333/trac/nst/`

1. The criterion `:perf` might be extended to provide (implementation-dependent) checks on memory limits as well as time limits.

2. Some sort of timeout mechanism — perhaps implementation-dependent — could be helpful for measuring correctness via termination in more complicated algorithms.

C.1 Removed features

Some features of the NST 0.9 have not yet been reimplemented. Tell John if one is urgent for you.

C.1.1 Test group documentation

The `def-test-group` form should allow group documentation:.

```
(:documentation DOC-STRING)
```

C.1.2 Alternate fixture definitions

The `def-capture/restore-fixtures` declaration binds `nil` to a collection of variables in the extent of associated test groups. This form is useful when hiding some developers' state from tests intended to simulate a non-development environment.

def-capture/restore-fixtures *name variables* `&key` *documentation*

C.1.3 Anonymous fixture sets

It is not necessary to name fixture sets; they may be given anonymously in any situation where a fixture set name is allowed. The syntax of anonymous fixture sets is:

`(:fixtures` *(name form)*$^+$ `)`

C.1.4 The `:with` check

This check assumes that the next item is a list, whose contents are expanded into the methods-and-forms. This check is especially useful with the `def-test-criterion` command below.
Syntax: `(:with` *criterion*`)`
Example: The following two `def-test`s are equivalent.

```
(def-test seq1
    (:seq (:predicate symbolp) (:eql 1) (:symbol d))
```

```
    '(a 1 d))
(def-test with-seq1
    (:with (:seq (:predicate symbolp) (:eql 1) (:symbol d)))
    '(a 1 d))
```

C.1.5 Compilation deferral control

In earlier versions the `:defer-compile` switch on test definition allowed control over when an expression would be compiled.

C.1.6 In the runtime system

C.1.7 The `:summarize-scheduled` operation

If `BOOL` evaluates to non-null, then the runtime system will print a summary after running scheduled tests with `:run`, `:continue`, etc.
Syntax: `:nst :summarize-scheduled BOOL`

C.1.8 The `:summarize-single` operation

If `BOOL` evaluates to non-null, then the runtime system will print a summary after one-time test runs initiated by `:run-test`, `:run-group`, etc.
Syntax: `:nst :summarize-single BOOL`

C.2 Marking tests of interest for execution

NST allows tests to be marked for execution by the `:run` command. Tests can be marked by package or group, or as an individual test.

C.2.1 The `:p` operation

Indicates that all tests in a particular package should be run.
Syntax: `:nst :p PACKAGE`

C.2.2 The `:g` operation

Indicates that all tests in a particular group should be run.
Syntax: `:nst :g GROUP`

C.2.3 The `:t` operation

Indicates that the named test should be run.
Syntax: `:nst :t TEST`

C.3 Test definition

C.3.1 The `:defer-test-compile` operation

Sets whether tests defined subsequently should, by default, defer compilation of their forms until actually running the test. This feature is useful when debugging code involving macros, but changing this feature in the runtime system can lead to confusion. It is surely almost always the right thing to set this flag locally via def-test-group and def-test. In fact, this operation may be removed in a future version of the runtime system.
Syntax: `:nst :defer-test-compile BOOL`

C.3.2 The `:open*` operation

Multiple fixtures can be *opened* into the interactive namespace with a single command using the `:nst :open` command:
Syntax: `:nst :open` *FIXTURE-NAME*∗

In earlier versions, only one fixture could be given to `:open`; this is no longer true, and the `open*` command has been removed.

C.3.3 The `:open-used` operation

If `BOOL` evaluates to non-null, then opening a fixture will always also open the fixtures it uses. Default is t.
Syntax: `:nst :open-used BOOL`

C.3.4 The `:reopen` operation

If `BOOL` evaluates to non-null, then fixtures will be re-opened *e.g.* when required multiple times by opening different fixtures that use them.
Syntax: `:nst :reopen BOOL`

Index